הטכניון – מכון טכנולוגי לישראל
מעבדה במערכות הפעלה 046210
תרגיל בית מס' 1

תאריך הגשה: 01.07.2024, עד 23:55

Introduction

Your mission in this assignment is to implement a simple Message Passing Interface (MPI) between processes. Each process will be able to send and receive messages to and from other processes. A message is made of an array of chars and does not include a NULL terminator. Sending and receiving these messages will be done using a new set of system calls, which you will implement.

Recommended Background Materials

From the course website, in the “short information” section:

- System calls
- Processes and useful data structures

Working Environment

You will be working on a RedHat Linux virtual machine, as in exercise 0.

Compiling the Kernel

In this assignment you will apply modifications to the Linux kernel. Errors in the kernel can render the machine unusable. Therefore, you will apply the changes to a ‘custom’ kernel. The sources of the custom kernel can be found in /usr/src/linux-2.4.18-14custom. All files that you will work with are in this directory. This kernel has already been configured and compiled. Refer to exercise 0 for the exact steps of compiling the kernel.

For a smoother submission process and to minimize the risk of encountering unexpected issues, it is highly advisable to set up a distinct directory where you can place the modified files from the kernel source and any new files you've added. This will help you keep track of changes and ensure that you don't miss any crucial files.

Once you've made your modifications, you can compile the kernel by copying the modified and new files back to the kernel source directory and following the steps outlined in exercise 0.

Alternatively, there's a convenient option to use the "make_changed.py" script provided in the "pygrader" package. This script automates the process of copying the necessary files to the kernel source directory and also creates a backup of the original kernel source code. Having a backup ensures that you can always start fresh if needed.

To use the "make_changed.py" script, first, download the "pygrader" zip file and install the package following the instructions in the README or on the course website. After installation, simply run the "make_changed.py" script as you would with any other command. This will handle the necessary tasks for you.

WARNING: do NOT put your changes directory inside the kernel source directory (example:

`/usr/src/linux-2.4.18-14custom/my_changes`**)!!!**

Example of modifying a kernel include file and compiling with the changes:

```
mkdir ~/my_changes
cd ~/my_changes
mkdir -p include/linux
cp /usr/src/linux-2.4.18-14custom/include/linux/sched.h include/linux/
# ...
# edit include/linux/sched.h
# ...
# Build the new kernel with the changes from the current directory
make_changed.py .
# If we messed up, we could return to a clean copy of the kernel
make_changed.py --reset
```

Tips for faster compilation

The kernel can take over 10 minutes to compile. To make development and debugging easier, follow these tips:

1. If you use `make_changed`, it is recommended that you compile the kernel once without it (like in ex0). This will significantly reduce your compilation time.
2. Changing header (.h) files will increase compilation times, so change them as seldom as possible.
3. Following 2, use the `-s` parameter to `make_changed` copy only the changed files over the existing kernel sources, instead of starting from a fresh copy and copying everything. Don't do this for the first compilation, since it will not create a clean backup, or if you deleted files from your changes directory, as this may cause strange bugs.
4. You can compile and install the kernel while it's running – no need to boot to the non-custom kernel to compile the custom one

Detailed Description

MPI is an important OS primitive that is used for communicating between programs. In this exercise you will implement a simple version of MPI, which allows basic sending and receiving of messages. A message in this exercise is an array of chars. The protocol will be invoked using a new set of system calls that you need to implement:

1. **mpi_register**: Register for MPI communications
2. **mpi_send**: Send a message to another process
3. **mpi_receive**: Receive an incoming message

You are required to implement both the system calls and their wrapper functions (wrapper functions simplify the invocation of system calls from user space). Detailed description of the new system calls and their wrapper functions is given in the next section.

Each process will hold a queue of incoming messages that haven't been read yet. This queue has no size limit. After reading a message using mpi_receive(), the message is deleted from the queue.

It's recommended (but not mandatory) to put the queue of received messages in **task_struct**, which is the database that stores all the information about a process. Since there is no limit to the number of messages, the queue should be implemented though dynamic allocation. For this, it's recommended (but again not mandatory) to use the kernel's linked list mechanism (see the implementation in "include/linux/list.h" and its use in the kernel code).

New System Calls API

You should implement the following system calls. The API is from the user's point-of-view, not the kernel's:

1. **int mpi_register(void)**
    a. Description:
       Register for MPI communication. After this system call returns, the calling process can send and receive MPI messages. Calling this function more than once does nothing.
    b. Return value:
        i. On failure: -1
        ii. On success: 0
    c. On failure, **errno** should contain one of the following values:
        i. "ENOMEM" (Out of memory): Failure allocating memory (if your implementation requires dynamic allocation)

2. **int mpi_send(pid_t pid, char *message, ssize_t message_size)**
    a. Description:
       Send a message of size **message_size** to the process identified by **pid**.
    b. Return value:
        i. on error: -1
        ii. on sucess: 0
    c. On error **errno** should contain one of following values:
        i. "ESRCH" (No such process): Process **pid** doesn't exist
        ii. "EPERM" (Operation not permitted): Either the sending process or **pid** isn't registered for MPI communication
        iii. "EINVAL" (Invalid argument): **message** is NULL or **message_size** < 1
        iv. "EFAULT" (Bad address): Error copying **message** from user space
3. **int mpi_receive(pid_t pid, char* message, ssize_t message_size)**
    a. Description:
       Check if the current process has a message from process **pid**. If there is, copy it to the buffer **message** with size **message_size** and delete it from the incoming messages queue. Messages are processed in the order they were received. If the message is longer than **message_size**, copy only the first **message_size** bytes and delete the message from the queue.
    b. Return value:
        i. on failure: -1

   ii. on success: The size of the string copied to **message**.

   c. On failure **errno** should contain one of following values:
      i. “EPERM” (Operation not permitted): The current process isn’t registered for MPI communication
      ii. “EINVAL” (Invalid argument): **message** is NULL or **message_size** < 1
      iii. “EAGAIN” (Resource temporarily unavailable): No message found from **pid**
      iv. “EFAULT” (Bad address): error writing to user buffer

4. On process creation (fork):
   a. If the parent process is registered for MPI communication, the child process will also be registered and have an empty message queue.
5. On process termination:
   a. The incoming message queue and incoming messages will be deleted and their memory freed.

Notes:

- If there are multiple errors that can be returned, you may return any of them.
- You may always return ENOMEM if your implementation requires dynamic allocation and it failed
- Remember that messages don’t include a NULL terminator, so make sure to add one yourself in your tests if you plan to print them using `printf("%s")`.

Your wrapper functions should follow the example in the next page (note: this is an example from a previous HW):

```
int add_message(int pid, const char *message, ssize_t message_size)
{
        int res;
         __asm__
        (
                "pushl %%eax;"
                "pushl %%ebx;"
                "pushl %%ecx;"
                "pushl %%edx;"
                "movl $243, %%eax;"
                "movl %1, %%ebx;"
                "movl %2, %%ecx;"
                "movl %3, %%edx;"
                "int $0x80;"
                "movl %%eax,%0;"
                "popl %%edx;"
                "popl %%ecx;"
                "popl %%ebx;"
                "popl %%eax;"
```

```
                    : "=m" (res)
                    : "m" (pid) ,"m" (message) ,"m"(message_size)
            );

            if (res < 0)
            {
                    errno = -res;
                    res = -1;
            }
            return res;
    }
```

This code uses inline assembler to call the 0x80 interrupt. Explanation:

1. **"pushl %%eax;"…**: Store any used registers in the stack.
2. **"movl $243, %%eax;"**: Store the system call number in register **eax.**
3. **"movl %1, %%ebx;"...**: Store the first parameter of the function in register **ebx.**
4. **"int $0x80;"**: Invoke the 0x80 interrupt.
5. **"movl %%eax,%0;"**: Store the return value (**eax**) in the output variable **%0**.
6. **"popl %%edx;"**: Pop back the stored registers.
7. **: "=m" (res)**: Map the output variable to variable **res**.
8. **: "m" (pid) ,…**: Map the input parameter **%1** to variable **pid**.

You can read more about inline assembly in the following link.

The wrapper functions should be stored in a file called "mpi_api.h".

The new system calls should use the following numbering:

System call	Number
mpi_register	243
mpi_send	244
mpi_receive	245

Useful Information

- You can assume that the system is with a single CPU.
- More on system calls can be found in the "Understanding the Linux Kernel" book.
- Use **printk** for debugging (see link). It is easiest to see **printk**'s output in the textual terminals: Ctrl+Alt+Fn (n=1..6). Note, since you are using the VMplayer you might need to press Ctrl+Alt+Space, then release the Space while still holding Ctrl+Alt and then press the required Fn.
- Use **copy_to_user** & **copy_from_user** to copy buffers between User space and Kernel space (see link).

- You are not allowed to use **syscall** functions to implement code wrappers, or to write the code wrappers for your system calls using the macro **_syscall1**. You should write the code wrappers according to the example of the code wrapper given above.

Testing Your Custom Kernel

You should test your new kernel thoroughly (including all functionality and error messages that you can simulate). Note that your code will be tested using an automatic tester. This means that you should pay attention to the exact syntax of the wrapper functions, their names and the header file that defines them. You can use whatever file naming you like for the source/header files that implement the system calls themselves, but they should compile and link using the kernel make file.

Submission Procedure

1. You should submit through the Moodle website (**Only one** submission per pair).
2. You should submit one zip file containing:
    a. All files you added or modified in your custom kernel. The files should be arranged in folders that preserve their relative path to the root path of the kernel source, i.e:
    ```
    zipfile -+
             |
             +- submitters.txt
             |
             +- mpi_api.h
             |
             +- kernel/ -+
             |           |
             |           +-...
             |
             +- include/ -+
             |            |
             |            +-...
            ...
    ```
    b. The wrapper functions and struct file "mpi_api.h".
    c. A file named "submitters.txt" which lists the names, **emails** and IDs of the participating students. The following format should be used:

    ploni almoni ploni@t2.technion.ac.il 123456789
    john smith john@gmail.com 123456789

    Note that you are required to include your email.

Emphasis Regarding Grade

- Your grade for this assignment makes 35% of final grade.
- Pay attention to all the requirements including error values.

- Your submissions will be checked using an automatic checker, pay attention to the submission procedure. Submission error will be penalized.
- You are allowed (and encouraged) to consult with your fellow students but you are not allowed to copy their code. You can share tests.
- Your code should be adequately documented and easy to read.
- Wrong or partial implementation of system calls might make them work on your computer, but not on others. Therefore, it is recommended to verify that your submission works on other computers before submitting.
- The kernel is sophisticated and complex. Therefore, it is **highly important** to use its programming conventions. Not using them might cause your code and **other kernel mechanisms** to malfunction. **Note that failing to do so might harm your grade.**
- Obviously, you must free all the dynamically allocated memory.